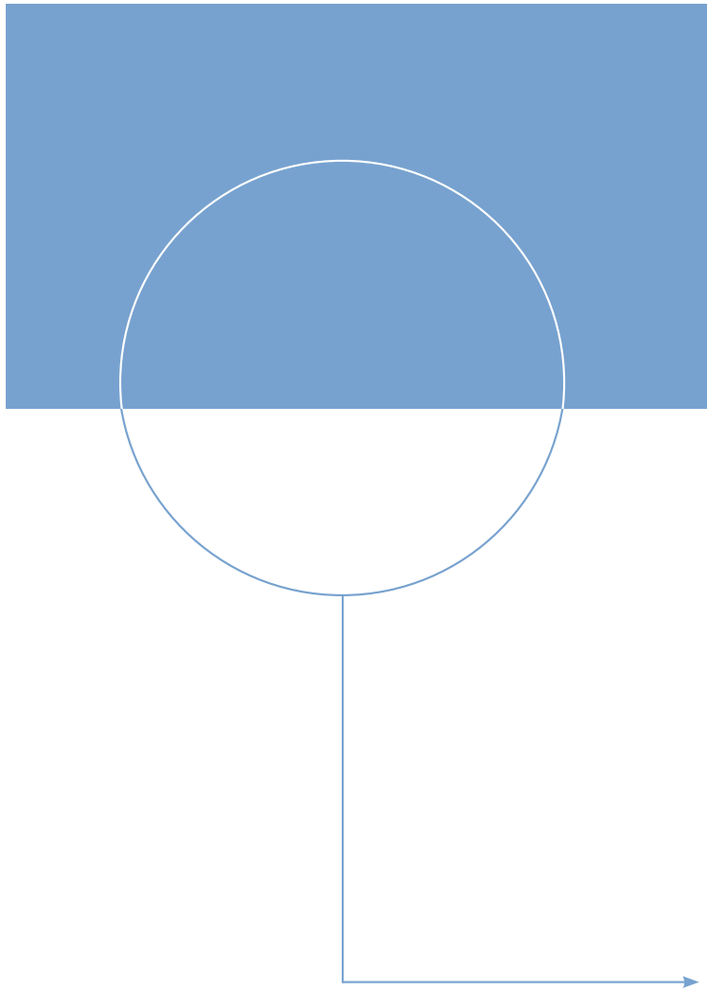


Master's thesis

2019

Alf Martin Eggan, Karl Andreas Eggan

NTNU
Norwegian University of
Science and Technology
Faculty of Information Technology and Electrical
Engineering
Department of Computer Science

Alf Martin Eggan
Karl Andreas Eggan

Estimating Tail-Latency of Latency-Sensitive Workloads

June 2019

NTNU
Norwegian University of
Science and Technology

NTNU
Norwegian University of
Science and Technology

NTNU

Norwegian University of
Science and Technology

Estimating Tail-Latency of Latency-Sensitive Workloads

Alf Martin Eggan

Karl Andreas Eggan

Computer Science

Submission date: June 2019

Supervisor: Rajiv Nishtala

Co-supervisor: Magnus Själander

Norwegian University of Science and Technology

Department of Computer Science

Problem Statement

In a typical server system, there are usually latency-critical workloads which are sensitive to user demands. These latency-critical workloads have a client-server interaction, where the client will send some request, and the server side application will have to respond within a given time frame. This time frame is typically referred to as Quality of Service (QoS) target. A higher number of responses within this time frame results in a higher quality of the service. The question is how to maximize the number of times the QoS target is met, subject to maximizing the energy efficiency or resource utilization (as a metric of Instructions per cycle, for instance).

To solve this problem, the following should be done:

1. Select one application from the Cloudsuite benchmark suite [1], and identify the type of queries generated by this application. ☐
2. Design a way to collect these metrics. There are two options: ☐
   (a) Instrument the specific workload and write the queries periodically to a log-file.
   (b) Collect metrics from the network interface card (NIC), as all requests will have to pass through it.
   The second approach is more generalizable.
3. Based on the current server system configuration (e.g., frequency of cores, number of cores, memory bandwidth, etc.), identify why certain queries take longer to finish. ☐
4. Based on the knowledge gathered from the previous points, present a prediction model that will estimate the QoS target. ☐
5. Build a prediction model that will be based on a neural network function approximator or some sort of mathematical formulation. ☐

If time permits:

1. Select one or more application and follow the same procedure as above. ☐
2. Build a scheduler that will meet this QoS target. ☐

Supervisor: Rajiv Nishtala
Co-supervisor: Magnus Själander

Abstract

Many of the workloads running on data center servers are latency-sensitive, and demand satisfactory user experience, making it challenging to reduce power consumption due to stringent tail latency targets. Task management mechanisms are crucial mechanisms to reduce energy usage subject to meeting tail latency targets. This thesis introduces Heimdall, a solution based on simple machine learning models (random forests, and support vector machines) to understand the correlation between cores, dynamic voltage and frequency scaling (DVFS), the load of the workload and latency. Our experimental results for Memcached performed on an Nvidia Jetson TX1 board show that Heimdall reduces energy consumption over two state-of-the-art task management solutions: Hipster and Heracles by 2% and 16% respectively, while improving times the tail-latency targets is met by 11.2% and 6.1%.

Sammendrag

Mange av arbeidsoppgavene som behandles i dagens datasentre er følsomme for forsinkelse, og krever at brukere har en tilfredsstillende opplevelse, noe som gjør energibesparing til en utfordring, som følge av strenge krav til forsinkelse. Oppgaveplanleggere er avgjørende for å redusere energiforbruket samtidig som disse kravene møtes. Denne masteroppgaven introduserer Heimdall, en løsning basert på enkle maskinlæringsmodeller (random forest og support vector machine) for å forstå korrelasjonen mellom kjerner, dynamisk spennings -og frekvensskalering (DVFS), arbeidsbelastning og forsinkelse. Våre eksperiment er gjennomført på et Nvidia Jetson TX1 brett, med resultater som viser at Heimdall reduserer energiforbruket med henholdsvis 2% og 16% i forhold til eksisterende oppgavebehandlere som Hipster og Heracles, og møter kravene for forsinkelse henholdsvis 11.2% og 6.1% oftere.

Preface

This project was done during the spring of 2019 as a final step towards our Master of Technology degree. During our work we have been supervised by PhD Rajiv Nishtala. We want to thank him for providing good and insightful discussions, giving valuable suggestions and ideas, and pushing us a bit extra when needed. We are also grateful that he has been patient and understanding.

We also want to thank our family for supporting us throughout the project.

Alf Martin and Karl Andreas Eggan, June 2019.

Contents

List of Figures

List of Tables

List of Abbreviations

API Application Programming Interface.

CDF Cumulative Distribution Function.

CPU Central Processing Unit.

DVFS Dynamic Voltage Frequency Scaling.

IPv4 Internet Protocol version 4.

ISN Initial Sequence Number.

MDP Markov Decision Process.

NIC Network Interface Card.

PAAE Percentage Absolute Average Error.

QoS Quality of Service.

RAM Random Access Memory.

RAPL Running Average Power Limit.

RF Random Forest.

RL Reinforcement Learning.

RPS Requests Per Second.

SVM Support Vector Machine.

TCP Transmission Control Protocol.

UDP User Datagram Protocol.

Chapter 1

Introduction

According to a report from 2014 [2], the energy consumption from US data centers, in 2013, was estimated to be 91 TW h. This matches the power consumption of all the households in New York City for two years. From an emission perspective this amount of energy is equivalent to 150 million metric tons of carbon pollution generated by coal-fired power plants.

Traditionally, power saving has been performed with focus on long-term performance, suited for batch jobs [3]. However, in recent years, there has been a rise of web-services with needs of rapid response times; latency-critical workloads. Latency-critical workloads are common workloads being processed on today's data center servers. These workloads are typically generated by popular web applications such as search, social networking, web mail, online maps, and automatic translation [3, 4].

It is quintessential for the aforementioned services to be responsive and provide quick responses. A study conducted in 2009 found that a server-side delay of more than 400 ms for page-rendering negatively affects user experience and advertising revenue [5]. Dean and Barasso demonstrate in Tail at Scale [4] that keeping latency low is a challenge because latency-critical applications usually operate on large scale data sets that may be fanned out over thousands of nodes. This signifies that when a user is sending a query, the result returned is combined of results from several individual nodes. To exemplify, having one slow node out of 100, a user request aggregating the results from these nodes will have a 63% chance of being slow. This urges the importance of tail-latency as a metric, the latency for which a percentage of all requests are within, e.g. the 95^{th} percentile. Thus, for the service to feel responsive, it is important that requests complete within a strict quality of service (QoS)-target, typically 95^{th} or 99^{th} percentile of the latency distribution. It is essential that these constraints are met in order to deliver a good and acceptable user experience.

To meet these requirements, data centers must be scaled to process requests within

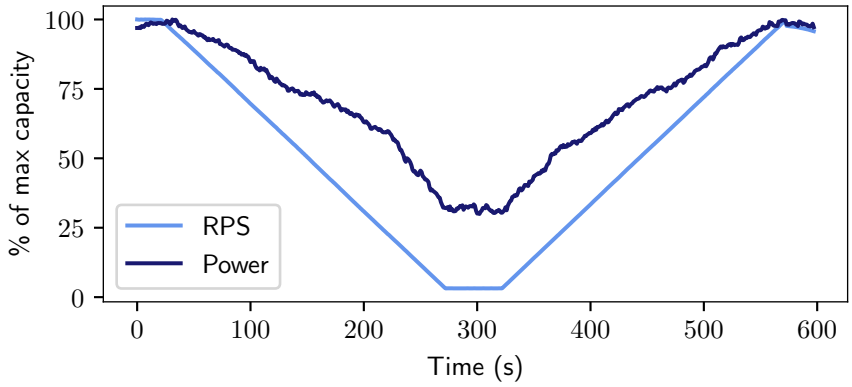


Figure 1.1: Power proportionality gap of Memcached on Jetson TX1

the QoS-target at maximum load. It is known that the use pattern of services like Google follow a diurnal pattern [3, 6], with load varying from 5% to 80% throughout the day. In addition, traffic spikes might occur due to uncommon events, like for instance Black Friday [7]. This leaves the servers under-utilized at times of lower load. Despite receiving a lower load, the energy consumption does not correlate linearly with the utilization, known as the power proportionality gap.

Figure 1.1 illustrates the power proportionality gap on our setup, which is described in section 4.1.1. We dedicate all cores, running at maximum DVFS, to Memcached, with the load varying over time. These results show, that even running at an embedded system utilizing power saving c-states, at 10% load, the power consumption is at 30%. This is the same trend that we see in one of Google's data centers, where the cluster draws 70% of peak power at 30% utilization [3].

With the traditional power scheme mechanisms being rendered inefficient for the demands of latency-critical workloads [3], new methods are being developed. Previous works [3, 6, 8–14] have presented various ways to address power management of servers running latency-critical workloads. At least two approaches have emerged for power saving, of which one is to use "optimal" system resources, in order to save power. The second is collocating other workloads alongside the latency-critical workload, in order to increase utilization of the resources available.

However, current task management policies are based either on the current load (in requests-per-second (RPS)) or tail latency, which may result in an increase in energy usage due to over-provisioning resources to ensure that tail latency is met. We demonstrate that this results in undesirable ping-ponging effects across mapping decisions for minor changes in load/latency.

To tackle the aforementioned problem, we design Heimdall, a task manager using simple machine learning models to estimate tail latency as a function of current load, core mapping and DVFS, and using this to drive task management decisions. The

machine learning models deployed are: random forests (RF), Support Vector Machines (SVM) and reinforcement learning (RL).

Specifically, first, we demonstrate that Memcached [15] load generators, in particular Cloudsuite [1] and Mutilate [16], exhibit a high degree of error when reporting important statistical information such as tail latency. We design a sniffer that allows for capturing of network packets between the client and server of a latency-critical workload, and for the extraction of characteristics that may aid in prediction of the tail-latency.

Next, we demonstrate that tail latency can in fact be estimated with a high accuracy using a small random data set drawn randomly from a sample set consisting of combinations of core and DVFS and load combinations.

Finally, we show that these models can be used to make effective scheduling decisions that out-perform state-of-the-art task managers such as the Linux scheduler, Hipster [8] and Heracles [6].

Our results show that, we improve over the Linux scheduler, Hipster and Heracles in meeting QoS by 25%, 11.2% and 6.1%, respectively, while reducing energy consumption by 1.2%, 2% and 16%, respectively.

The report is structured as follows. Chapter 2 gives motivation for the research and present background information and review of work related to latency-critical workloads. In Chapter 3 we describe work developed during the research. Chapter 4 presents our experimental setup and experiments, and thereafter the results obtained from these experiments. Finally, chapter 5 concludes the project.

1.1 Thesis Timeline

Table 1.1 shows the development and focus we have had throughout the work of this thesis, and tries to explain where time was spent.

Initially for this project we had a stronger emphasis on capturing per-packet characteristics, for instance whether a request is a get or a set, and using it to make more precise estimates. As an additional feature to this, we wanted to make a completely workload agnostic sniffer, that was able to automatically profile a workload and extract these characteristics. After constructing the code with that in mind, and researching various methods of blind packet analysis and pattern recognition, we found it to be out of scope, and that it requires much more effort than it would benefit the thesis.

A considerable amount of time was also used configuring and setting up the load generators. We initially started with Cloudsuite's Memcached load generator, and found it to be too unstable to function well with reinforcement learning in April. Due to this, we had to setup and test several other load generators for Memcached. Effort

Date	Description
15 January 2019	Signe master contract
January 2019	Read up on related work
January 2019	Setup Memcached and Jetson TX1
Januray 2019	Setup and experiment with Cloudsuite
February 2019	Build and test sniffer
March 2019	Add application protocol profiling to the sniffer (unsuccessful)
March 2019	Implement reinforcement learning
April 2019	Setup and experiment with Mutilate
April 2019	Test various other load generators (Treadmill, Masstree, Mcperf)
April 2019	Setup web search, both server and load generator
May 2019	Implement estimations using conjugate gradient based on ratio of processed request and outstanding requests (unsuccessful)
May 2019	Implement SVM and RF
May 2019	Run scheduling experiments on Memcached
28 May 2019	Write report
June 2019	Experiment on web search (incomplete)
11 June 2019	Deliver report

Table 1.1: Thesis timeline

was also put into building a web search index and setting up web search, as we had plans to do experimental testing with that workload as well.

Chapter 2

Background and Literature Review

In this chapter we look into the closely related research to our topic. We also explain the different machine learning models employed in this work, as well as explaining network packet capturing.

2.1 Energy Efficiency

The power proportionality gap is a major problem for energy efficiency in modern data centers. Due to static power, lower load does not necessarily result in reduced energy usage. In order to improve power proportionality, a server should be used to its maximum utilization. Multiple techniques have been used to manage resources in data center servers, and we will describe two of them; core mapping and DVFS.

2.1.1 Resource Scaling

DVFS is a technique by which the processors' power consumption can be controlled by varying the voltage/frequency dynamically [17]. The DVFS state of the processor is currently termed "target frequency". The target frequency is controlled using CPU governors. The three predominantly used CPU governors are: *ondemand*, *userspace*, and *performance*. The governor ondemand sets the DVFS state based on the current processor utilization, as seen using the tool "top". If a threshold limit is exceeded the frequency is scaled up to maximum. The governor userspace explicitly allows the user the define the processors' DVFS state, and will not scale the DVFS state based on requirement. The governor performance sets the highest DVFS state.

Core mapping is used to allocate a specified amount of resources (i.e., cores) to a workload by binding the workload to core(s). This is achieved by using the Linux command line tool "taskset" or "sched_setaffinity". Core mapping is useful

when a workload is compute bound, as it does not waste any compute cycles stalling for memory [18].

For the rest of the thesis, we use DVFS [3, 6, 8, 9, 11, 14, 19] and core mapping [6, 8, 20] as mechanisms to control the resources (i.e., cores and the DVFS state of those cores) for each user specified workload running on the system.

2.1.2 Collocation

Large-scale batch workloads, such as file backup and offline image processing, are not latency-critical and have no QoS constraints. In many data centers it is desirable to run both latency-critical and batch workloads to increase server utilization at periods of low load, due to the power proportionality gap. However, collocation of workloads is not trivial, and prior work [6, 8, 10, 11, 13, 14] have addressed several challenges. This is an important approach to handling the inefficiencies of the power proportionality gap. As collocation was not exploited in this thesis, we stop the discussion here.

2.2 Machine Learning Models

In our work, we approach the task management problem by estimating the tail latency, and perform core mapping and DVFS decisions based on the predicted results. These predictions are estimated by machine learning models. We investigated three types of machine learning models: Support Vector Machine (SVM) regressor, Random forest, and reinforcement learning.

SVM Regressor [21] is a generalisation of SVM to solve regression problems;

Random Forest Regressor [22] is a supervised learning algorithm that assembles multiple decision trees to predict a decision for a given mapping function;

Reinforcement learning is a framework for an autonomous agent to interact in a dynamic environment and learn the optimal policy based on reward earned at each state and action using the exploration-exploitation dilemma. In a reinforcement learning problem, the agent learns the optimal policy by interacting with the environment using the Markov Decision Process (MDP). Q-learning is the most popular form of reinforcement learning and is capable of achieving impressive results for a wide range of problems (e.g., playing Go, Atari, etc). A single step of the Q-learning problem is given by

$$\mathscr{R}_{new}(s_t, a_t) \leftarrow (1-\alpha) \cdot \mathscr{R}(s_t, a_t) + \alpha \cdot (r_t + \gamma \cdot \max_a \mathscr{R}(s_{t+1}, a)) \tag{2.1}$$

The model maintains a lookup table, $\mathscr{R}(s, a)$, of estimated discounted rewards for an action, $a \in A$, in state $s \in S$. At a discrete time step, t_n, in state s_n, the action taken will be the one estimated to give the greatest total, discounted reward: $a_n = argmax_a(\mathscr{R}(s_n, a))$. Upon receiving the reward, r_n, $\mathscr{R}(s_n, a_n)$ is updated as illustrated in equation 2.1, based on both the old and new state, s_n and s_{n+1}, as well as the reward, r_n. The learning rate is denoted as $\alpha \in (0, 1]$.

2.3 Workloads

2.3.1 Load Generators

In a real world scenario, data center servers are loaded by requests from individual users. However, when conducting a performance evaluation this is not a practical option. To simulate a request stream from a client to a server, a dedicated program, called a load generator, is used. The load generator resides on a client machine and is periodically constructing and sending requests to achieve a desired server load, commonly measured in RPS.

Every load generator has one of two control loops. It may either be closed loop or open loop. A closed loop load generator will have blocking requests, meaning that new requests are held back until the previous request on the connection is responded to. This entails that the maximum number of outstanding requests may never exceed the number of connections. An open loop may send requests to match target load, regardless of response.

In this work, we have experimented with Mutilate [7] and Cloudsuite's load generator [1]. Mutilate is closed-loop, while Cloudsuite's load generator is open-loop.

2.3.2 Memcached

Memcached [15] is a commonly used in-memory key-value store intended to speed up dynamic web applications. It is extensively used as a caching system by large-scale services, such as for Facebook [23] and Twitter [24], for latency-critical workloads. It has also been extensively studied in academia [3, 6, 8, 9]. By storing data objects in faster memory, Memcached manages to cache frequently used data for quick access. To alleviate database load, web servers will typically try to read values from Memcached before accessing the slower backend databases.

Memcached has two protocol modes: ASCII string queries and binary. We use the binary mode as it is more modern [25]. The layout of the binary header is seen in figure 2.1. The binary packet header is always 24 bytes. There are three more fields in

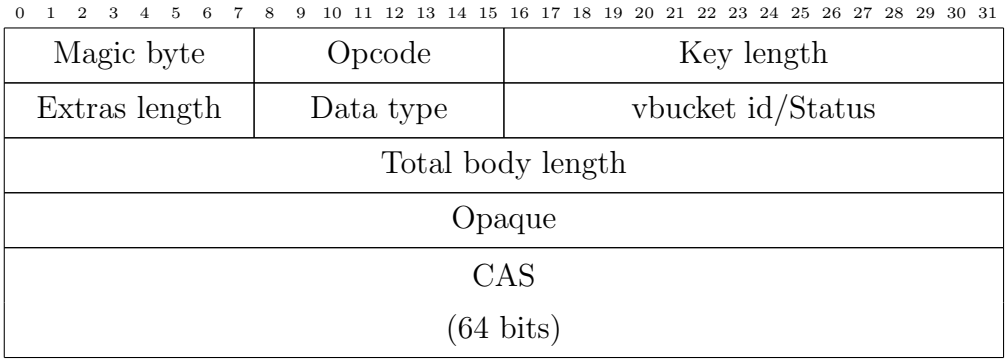


Figure 2.1: Memcached binary protocol layout

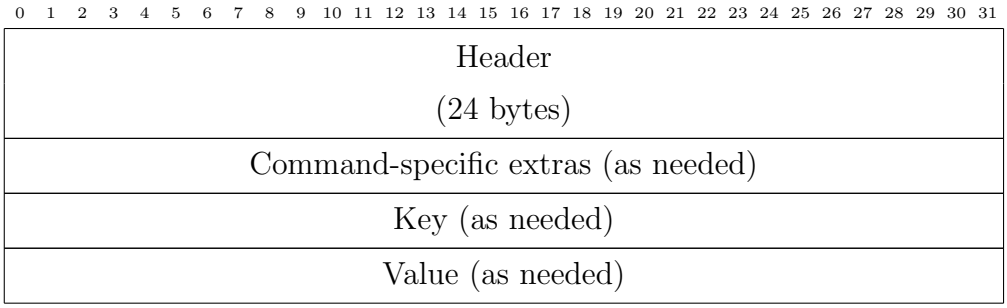


Figure 2.2: Memcached packet structure

a Memcached packet, as seen in figure 2.2, all sized as needed for each request/response.

In the header, the magic byte has two purposes. (1) it indicates whether it is a request or response, and whether it is from the server or from the client. (2), it will behave as a version control, as the byte values change for each version. In order to know which operation the query is issued, the second byte contains the opcode. There are in total 146 different opcodes, but most are not used by the load generators tested in this work. The opcode values that are used may be seen in table 2.1. When sending several, concurrent requests, the client must be able to recognize which request is being responded to when the order of responses is not maintained. The opaque field is generated when a request is sent, and is echoed back by the corresponding response from the server, enabling the client to match request and response.

Byte value	*Query type*
0	Get
1	Set

Table 2.1: Memcached binary opcodes

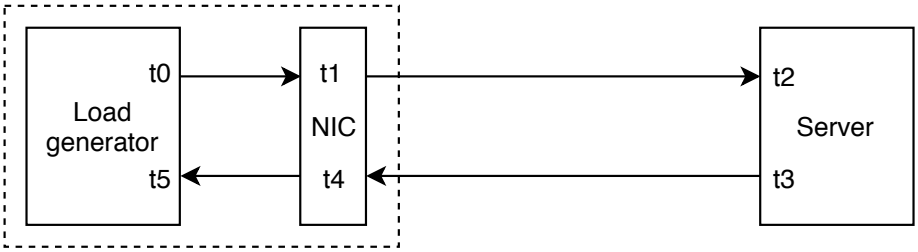


Figure 2.3: Request flow between load generator and sniffer

2.3.3 Web Search

We use the benchmark provided by Cloudsuite for Web Search [26]. The server relies on the Apache Solr [27] search engine framework.

The load generator from Cloudsuite encodes the requests with *chunked transfer encoding* [28]. This way, each request may be split into chucks and transferred over several packets, in a series concluded by a packet only containing the byte sequence 0x0d, 0x0a, 0x30, 0x0d, 0x0a, 0x0d, 0x0a. Each chunk begins by denoting the length of the chunk in hexadecimal, followed by 0x0d, 0x0a, then the content of the chunk, and concluded by 0x0d, 0x0a.

2.4 Networking

Current state-of-the-art task managers [6, 8, 9] depend on application performance metrics as input to handle latency-critical workloads. These metrics have to be collected, and provided to the scheme. One approach to obtain these metrics is to instrument the specific workload and write the queries periodically to a log-file. This method requires manipulation of the load generator source code, which requires extra effort, and is not generalizable. Additionally with workloads possibly generating tens to hundreds of thousands requests per second, per-query precision will be difficult to register due to slow I/O-operations. Another possibility is to exploit that all requests will have to pass through the network interface card (NIC), as shown in figure 2.3. This way we can collect the application-performance metrics in a load generator independent way. Figure 2.3 illustrates the request flow between the load generator and the server. With indices in increasing order, a request is sent from the load generator process (t_0), through the NIC (t_1), and then arrives the server (t_2). The response follows the same route in reverse, starting at the server (t_3), entering the client's NIC (t_4), before finally reaching the load generator process (t_5).

Previous work [29] has shown that queuing effects in the client can occur when deploying workloads with high request rates. For user experience, the end-to-end latency

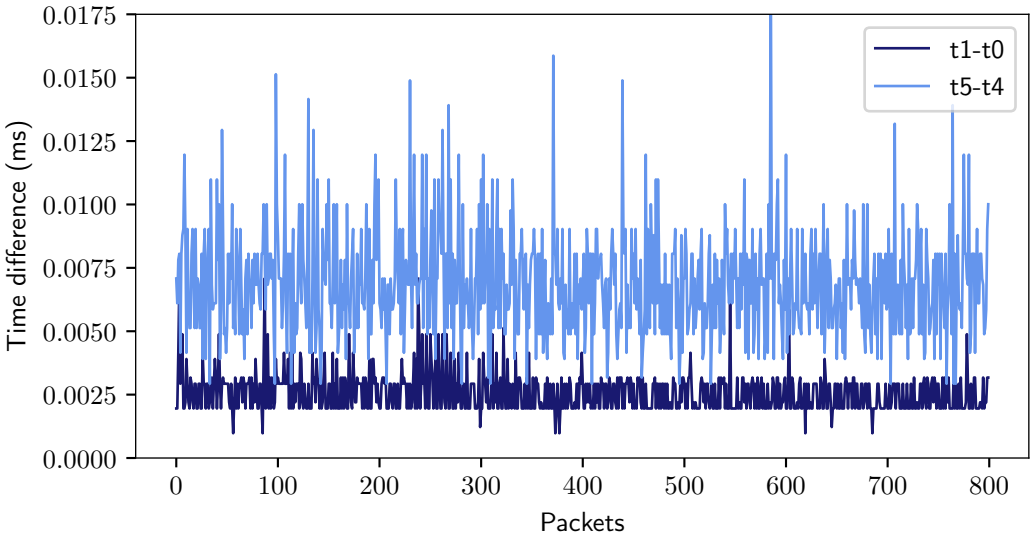


Figure 2.4: Time difference between timestamp from load generator and NIC, measured per packet

is what affects the performance. This is the time from when the user sends a request until the response is received. However, for load generators, the client-side queuing will bias the latency measurements. Figure 2.4 shows two graphs that illustrate the time difference between the load generator and the NIC for requests sent from the load generator (t_1-t_0), and responses received by the load generator (t_5-t_4). We observe that the difference between t_1-t_0 is less than between t_5-t_4, due to queuing latency.

We demonstrate in section 4.1.2 that the client-side latency for CloudSuite [1] increases as the server utilization grows. Collecting the metrics from the NIC directly, this bias is eliminated.

2.4.1 PCAP

In order to monitor the NIC, we use *pcap* [30], an API developed by "The TCPdump Group" that allows the user to capture network traffic. The library is implemented in C/C++, with support for other languages through various wrappers [31–33]. *pcap* is commonly used in packet capturing tools, like tcpdump [30] and wireshark [34]. Packets are time stamped when arriving the NIC. Along with the timestamp, the user is provided with the entire packet frame.

In this work, we will employ this library to provide our network packet sniffing tool with the network packets that are being exchanged between the server and client.

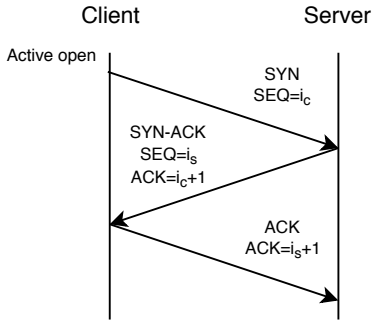


Figure 2.5: Three-way handshake to initiate a TCP connection

2.4.2 Transmission Control Protocol (TCP)

TCP [35], transmission control protocol, is a common protocol for the transport layer in the network stack. Its two most important properties, that sets it apart from other protocols like for instance the user datagram protocol (UDP), is loss less end-to-end packet exchange, and maintenance of packet order [36].

In order to keep track of the amount of data sent and successfully received over one connection, as well as the packet order, the TCP utilizes sequence numbers and acknowledgement numbers. At the initialization of a new connection, both the server and the client generate their individual initial sequence number (ISN), using an initial sequence number generator. The ISN may be an arbitrary number generated by the ISN generator. For simplicity in this example we denote the server ISN, i_s, and the client ISN, i_c. These are then being exchanged between the client and server in a *three-way handshake*, as seen in figure 2.5.

The client initiates the connection in what is called an *active open*. In an active open, the client sends a synchronize (SYN) packet containing i_c to the server. The server responds with a synchronize and acknowledgement packet (SYN-ACK). In this packet the server acknowledges the clients ISN with an ACK value of $i_c + 1$, and shares its own ISN. The last step in the handshake is the client acknowledging the ISN of the server, by sending an ACK value of $i_s + 1$. The setup is complete, and both the client and server can communicate.

During packet exchange, the sequence number of a packet is a counter of how many bytes (application layer data, when excluding the underlying protocols) has been sent from the sending side up to, but not including, that specific packet. The receiving side will acknowledge the amount of bytes received up to that point. With s being the sequence number, a being the acknowledgement number, and l being bytes sent until that point, the calculation of the sequence and acknowledgement number for the client is expressed in equation 2.2

$$\begin{aligned} s_c &= i_c + l_c \\ a_c &= i_s + l_s \end{aligned} \tag{2.2}$$

Reception order is ensured by enforcing monotonically increasing sequence numbers. If a packet has a lower sequence number than the previous, one of two things has happened; (1) the sequence number has already been received, and it is a retransmission, or, (2) it is a previously unseen packet received in the wrong order. Loss-less communication is ensured by validating that the sequence numbers are acknowledged.

For a packet sniffing tool this is useful because it connects sending and receiving packets. Specifically, in a closed-loop load generator, each connection has blocking requests, forcing it to wait for the previous request to be responded before issuing a new request. Using this, it is possible to match a client request with the server response, as the acknowledgement number from the server will be $s_c + payload$ of the request.

2.5 Related Work

Hipster [8] is a hybrid scheme that combines heuristics and reinforcement learning to determine mapping decisions based on the current load of the workloads. However, it becomes limited when scaling across large server systems, as it stores a large look-up table containing values for all state-action combinations.

Heracles [6] is a heuristic feedback-based controller that manages four isolation mechanisms to meet the QoS-target. However, as it is designed to enable a latency critical workload to be collocated with batch workloads, it schedules to higher frequencies even at low load.

Adrenaline [9] leverages DVFS at a finer granularity than what has traditionally been done. It uses application-specific information, like for example query type(get/set) for Memcached, to pinpoint and boost long running queries. This makes it reliant to prior knowledge about the application. Furthermore, it is using fine-grain voltage boosting techniques that requires simulated hardware to achieve DVFS scaling with nanosecond precision.

Pegasus [3] uses a feedback-based controller designed to improve energy proportionality of data center servers running latency-critical workloads. It operates as a state-machine and adjusts DVFS via RAPL registers, in response to measured tail latency, and adapts to diurnal loads. However, it does not respond quickly to short-term variability.

It is also argued in Pegasus [3] that processor utilization alone (as used by ondemand) is a bad metric and cannot be used to reduce energy consumption subject to meeting the QoS targets of the latency-critical workloads. This is because, ondemand

governor controls the DVFS state based only on the CPU utilization, whereas these workloads are also network bound.

Smooth Operator [10] is a framework that analyzes temporal heterogeneity of power consumption patters, and derives highly power efficient service placement. The power infrastructure of many data centers is built as a hierarchy, where servers only draw power from leaf nodes, leaving head room of the root unusable. In addition, they have to be provisioned for peak power usage. As each service has its own power pattern, placing complimentary services under the same power node decreases the power peak, creates a more stable draw of power under the same node, and achieves higher throughput in data centers without changing the existing power infrastructure.

Rubik [14] is a fine-grain power management scheme for latency-critical workloads. It uses queue length as a measure of instantaneous load and adjusts DVFS whenever the queue length changes. Longer queue waiting times result in higher DVFS setting.

Q-Clouds [13] is a QoS-aware control framework for tuning resource allocations between VMs to avoid interference effects. A multiple-input multiple-output model is built based on online feedback and describes the relationship between resource allocations and the QoS experienced by VMs. To utilize unassigned resources higher QoS-levels are provided consumers that are willing to pay for it.

Bubble-up [11] is a characterization methodology that enables accurate prediction of performance degradation when services are collocated. By measuring each service's sensitivity and pressure to each shared resource, e.g. shared last level caches, bandwidth to memory, etc., Bubble-Up detects complimentary services. A service with high sensitivity for a shared resource should not be placed with a service with high pressure to the same shared resource. However, this is a static method, and requires *a priori* knowledge about the workloads, it is designed to find two complimentary services, and does not scale to collocation of more than two services simultaneously, and finally, it is not able to adapt to phases and load changes during or across executions.

Amdahl's Law for Tail Latency [37] investigates how focus on tail-latency affects hardware design. Their findings suggest that heterogeneous systems perform better for moderate QoS-targets, and bigger cores are more favorable as serialization increases. In addition, when a service has low latency constraints, one should balance single-thread performance and request-level parallelism. In that case very weak cores may not handle variability in service time, and the single-thread performance of stronger cores is needed, while several weak cores are favored when a service is throughput bound.

Chapter 3

Heimdall

This chapter presents Heimdall, a task manager using simple machine learning models to estimate tail latency as a function of current load, core mapping and DVFS. It incorporates a dedicated sniffer to provide the task manager with application performance data. The goal is to make effective scheduling decisions that increases energy efficiency subject to meeting the QoS-target.

We divide our work into three parts. The first part concerns the sniffer. We explain how it is constructed and deployed. The second part revolves around the prediction models. The third part is combining the work from part one and two to make scheduling decisions.

3.1 Sniffer

The sniffer is the foundation of this work, as the task manager will build on top of it. Its main functionality is capturing the traffic between the client and server of interest, extract characteristics and details for each request, and finally present it to the overlaying mechanism, the task manager. It aims to have a modular, pipelined design, see figure 3.1. Each component has a specific, isolated role.

3.1.1 Packet Sniffer

The first component is the packet sniffer. This component captures packets on a network interface, and passes it to the packet extractor. Capturing is done using the

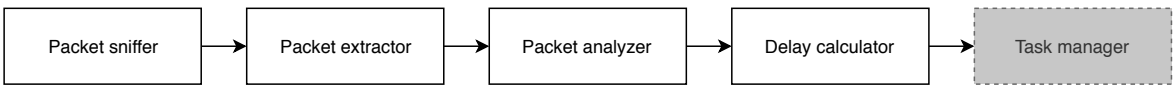


Figure 3.1: Architecture of the sniffer

libpcap API. Settings like what network interface to listen to, and filters on for instance ip address and port, is set in this module. If only packet counting, and/or header reading is of importance, one may also set a maximum buffer size. This allows the sniffer to only store the first n number of bytes in the buffer, discarding the remaining content of the packet. For instance, when analyzing Memcached, and only RPS and the ratio of get requests compared to set requests is of importance, the header contains all the information needed from a packet. By only collecting the first 80 bytes of a request (66 bytes of underlying protocols + 24 bytes of Memcached header), we get all the data required to calculate these metrics. This way, keys and values that may span hundreds of bytes do not have to be stored, and the memory requirements are reduced.

After the setup is done, the program enters an infinite loop. Whenever a new packet enters the NIC, a user defined callback is executed, with access to the bytes of the frame, including protocols down to the link layer, length of the packet and time stamp. This data is propagated on to the packet extractor.

3.1.2 Packet Extractor

When processing the data from the packet sniffer, the packet extractor dissects the packet content protocol by protocol down to the payload. We add support for IPv4 and TCP. Pointers to the underlying protocols are stored as corresponding **C** structs and passed to the packet analyzer along with the pointer to the payload. These structs, as well as the time stamp and packet length, are stored in the struct packet_meta_data_t (see Listing 3.1).

```
1  typedef struct {
2      ethernet_header *ethernet;
3      ip_header       *ip;
4      tcp_header      *tcp;
5      struct timeval  ts;
6      uint32_t        seq_next;
7      uint32_t        len;
8      uint32_t        total_len;
9  } packet_meta_data_t;
```

Listing 3.1: Layout of the structure packet_meta_data_t

3.1.3 Packet Analyzer

The first two steps of the pipeline, the packet sniffer and packet extractor, are application agnostic. The application specific analysis is performed in the packet analyzer. In order to make it adaptable to a wider range of applications, this component is mul-

tiplexing the data from the packet extractor to code specialized for the application. This can be expanded by the user by providing a dedicated function.

```
void handle_packet(void *p, packet_meta_data_t *m, const char *buffer,
    unsigned int l) {
    delay_calculator_t *d = (delay_calculator_t *)((sniffer_params_t*)p)
    ->args;

    if(d->application_id == 0) {
        // Call to custom analysis code
        analyze_custom(d, m, buffer, l);
    } else if(d->application_id == 1) {
        analyze_memcached(d, m, buffer, l);
    } else if(d->application_id == 2) {
        analyze_unknown(d, m, buffer, l);
    }
}
```

Listing 3.2: Example of protocol multiplexing

This is useful for instance when analyzing Memcached, as the analyzer is able to read out protocol specific traits, for instance the magic byte, opcode, opaque, etc. Although different workload protocols have different traits, some information can still be extracted in a protocol agnostic manner. Consequently, the packet analyzer implemented in our sniffer has a function for unknown protocols, seen in Listing 3.2 as analyze_unknown(). This function matches packets as described in section 2.4.2, using sequence numbers and acknowledgement numbers. The unknown protocol function determines whether a packet is a request or a response, as well as request length.

In analyze_memcached, the packets are matched on opaque as well as port. As described in Section 2.3.2, this is the field that gets echoed back from the server when a request is sent, and should be different for requests on the same connection.

When analyzing websearch, it is required to reassemble the packets for each individual request, due to the chunked encoding described in section 2.3.3. In order to achieve this, we maintain a table with an entry for each active connection, storing the packet data as they arrive until the terminating chunk is received. Only when the entire request is received, it is concatenated into one, single packet, the packet is considered reassembled, and packet analysis is performed. This allows us to calculate the packet length, as well as count requests per second for web search.

3.1.4 Delay Calculator

The core functionality of the delay calculator is calculating the tail-latency and similar statistics. It also manages these statistics at the time interval defined by the user. At

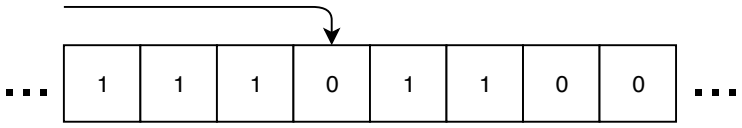


Figure 3.2: Naive storage

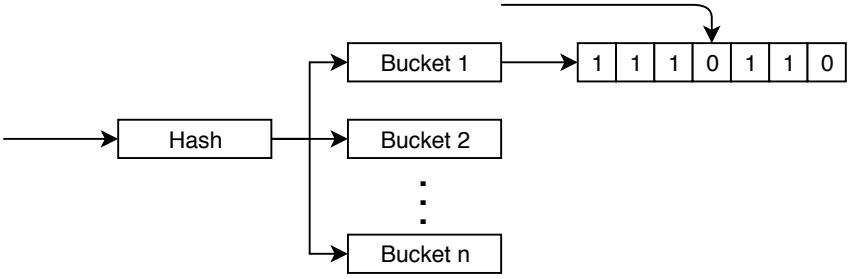


Figure 3.3: Hashed storage

this point in the pipeline, statistics such as load, tail-latency and workload specific metrics are refined, and may be printed to screen, or passed on to the task manager.

Two methods were implemented for storing requests. The first method, referred to as the naive method, is a 1-dimensional array. Each position in the array contains an entry, consisting of an active flag and meta-data about the request. At insertion of a new request, the delay calculator will search the array from index 0 to index l, where l is the length of the array, until a non-active entry is found. If all entries are active, the array will be reallocated to double size, and the new request will be stored at index $l_{new}/2 + 1$, where l_{new} is the new length of the array. When performing a lookup, the delay calculator searches the array from index 0 to index l, and compares the key and the port of the new request for each active entry. When an active entry with matching key and port is found, it is returned. If not found, *NULL* is returned. The search is illustrated in figure 3.2

The second method of storing requests embeds a hashing table to better cope with a higher number of simultaneous entries. While being viable at low loads, the naive method becomes inefficient as the load and number of outstanding requests increase. By using a uniformly distributed hash function, we can divide the requests into n buckets. As hash functions are $\mathcal{O}(1)$, this reduces the complexity of every search. Regardless, having an amount of buckets equal to the number of outstanding requests is wasteful, as the table must be allocated at start up. We therefore combine the hash table with a linked list, as illustrated in figure 3.3.

The key is stored in a 64-bit integer. The port is necessary as different connections may share the same key. When matching packets based on sequence numbers and acknowledgement numbers, as explained in section 2.4.2, each connection maintain

their own sequence numbers individually.

When a connection is closed with an outstanding request, the server has no medium through which it can respond. This will leave an unresponded entry in the delay calculator database that will never be neither looked up nor removed. Over time, this occupies memory, and gives erroneous measures of outstanding requests. To address this, a cleaning function is implemented, that at a set time interval removes all entries older than a predefined value. This value should be an order of magnitudes larger than QoS-target to avoid removing active requests.

3.2 Machine Learning Models

The data collected in the sniffer is passed to the machine learning models. The main goal of the machine learning models is predicting the tail-latency for a given state, by taking RPS, core allocation and DVFS as input.

3.2.1 Support Vector Machine and Random Forest

We use the *Scikit-learn* library [38] to implement SVM and RF. Both models require a labeled training set, and an offline training phase before predicting real-time samples from the sniffer.

3.2.2 Reinforcement Learning

We implement the reinforcement learning problem from scratch as we could not find any suitable libraries in C. This provided us with: (1) good insight into reinforcement learning, (2) flexibility to our code.

Our prediction problem is translated to a MDP as follows: the state, s_n, is the core and DVFS configuration, as well as RPS in the time interval t_{n-1} to t_n. RPS is quantized into buckets. Based on the state, the model selects an action, a_n, being its tail-latency prediction for time interval t_n to t_{n+1}. The set of actions, A, consists of ten estimation values. The estimated values are equally sized buckets in the range of the minimum latency to the QoS-target. All violations reside in the uppermost bucket. The reward for an estimation, r_n, is formulated as the negative, absolute error from the measured value, see equation 3.1.

$$r_n = -|l_{predicted} - l_{measured}| \tag{3.1}$$

In our task manager, we must predict the tail-latency for all states when scheduling, and then choose the best state. As all these are hypothetical states, and we do not

have an answer to them, simulated actions are done, that do not update the $\mathscr{R}$-table. The RL model is only updated for the state that is chosen by the task manager, and is in that way continuously learning and improving.

3.3 Scheduling

We exploit the output generated by the sniffer and the prediction model to build a task manager. The task manager is responsible for determining core mapping and DVFS settings for the server, in order to increases energy efficiency while meeting the QoS-target.

Figure 3.5 shows a high-level overview of Heimdall. It includes a QoS-monitor, a prediction module and a mapper module. The QoS-monitor, i.e. the sniffer, periodically collects statistical information about application-level metrics, such as RPS and query latency. These statistics, along with core and DVFS configurations, constitute the state. The statistics are passed to the prediction module and used by the machine learning model to predict the tail-latency for the next time interval. All possible combinations of core and DVFS configurations are sequentially fed into the prediction model along with the collected application-level data. As a result, tail-latency predictions for all possible states in the next time interval are known.

The predictions are evaluated before actuating the mapper module. Due to the power proportionality gap, it is beneficial to scale down the system resources as much as possible while meeting the QoS-target. Figure 3.4 demonstrates tail latencies at different loads and configurations, where bigger size of the scatter point indicates higher tail latency. We observe a correlation between scaling down system resources and an increase in tail latency. From this we instruct the latency evaluator to select the core and DVFS configuration that meets as close to the QoS-target without violation.

As we demonstrate in section 4.2, the prediction models tend to underestimate the latency. To compensate, we determine a slack value, s, from an empirical analysis, to approximately 10%. When evaluating the predictions for all the configurations, the slack point (QoS-target - s) will be used as target rather than QoS-target.

$$
\begin{aligned}
x' &= argmax_x(f(x)) \\
\text{f(x)} &= \begin{cases} l(x), & l(x) < l_{target} - s \\ l_{target} - l(x) - s, & else \end{cases}
\end{aligned} \tag{3.2}
$$

The configuration selection process is expressed in equation 3.2, where x is the state, $l(x)$ is the tail-latency prediction for state x, and l_{target} is the QoS-target. $f(x)$ is the evaluation function, expressing the score of a certain configuration. In short, it

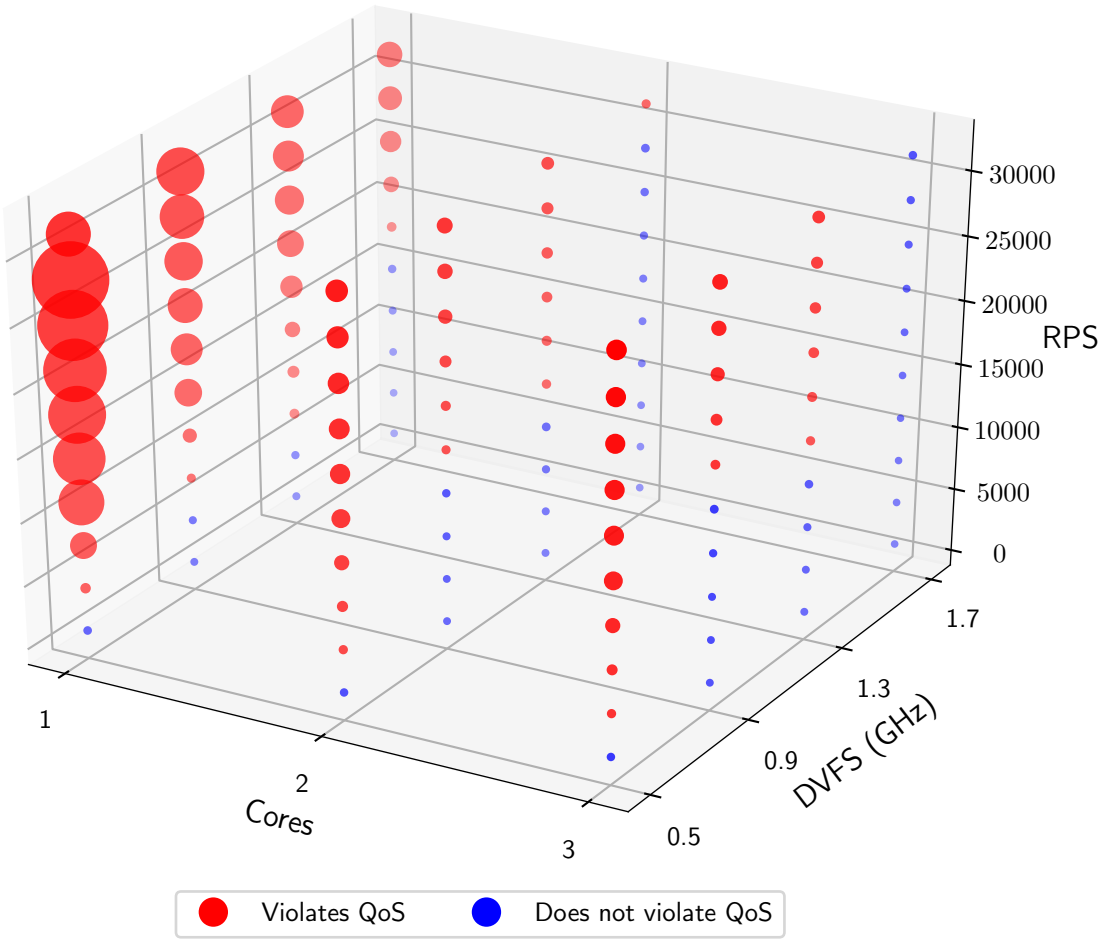


Figure 3.4: Latency at different DVFS/mapping configurations

selects the highest non-violating configuration. If all configurations violate the target, the configuration with the lowest latency is selected.

The latency evaluator propagates the most suitable configuration to the mapper module, which notifies the server to change core and DVFS configuration. Heimdall is operating on a dedicated node, and a communication protocol over TCP is developed to enable communication to the server nodes operating the workload. The protocol is using nine bytes. The first byte contains the binary flags, and the next 8 bytes contain the data. Only if flag seven is set, eight additional bytes are appended containing a timestamp, making it in total 17 bytes. The effects of the flags can be seen in table 3.1. This allows Heimdall to both set DVFS and core mapping, as well as read the current configuration and energy consumption.

On the node serving the workload, the DVFS is adjusted and the workload is allocated to cores according to the information received from the mapper module. Linux' *sched_set_affinity* system call is used to specify core allocations while for DVFS control, we select the *userspace* governor and adjusts the DVFS with *cpufreq*.

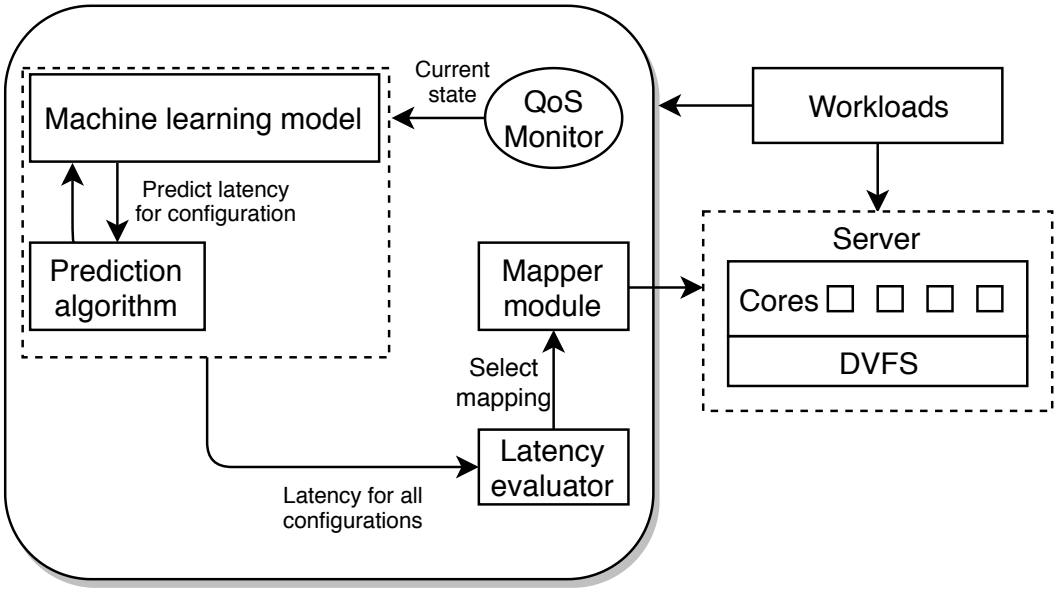


Figure 3.5: High-level overview of task manager model

Bit	*Effect*
0	Unused
1	Indicates successfull request
2	Indicates that request is a set
3	Indicates that 8-byte value is a double
4	Indicates that energy is requested
5	Indicates that core count is requested
6	Indicates that frequency is requested
7	Indicates that request appends an 8-byte timestamp

Table 3.1: TCP controller protocol

Chapter 4

Evaluation

This chapter covers our experimental setup, both in software and hardware, and the results of experiments. The main focus is on scheduling, but evaluations of the prediction models, as well as the sniffer, is also done.

4.1 Experimental Methodology

4.1.1 Experimental Platform

We perform the evaluation of Heimdall on an Nvidia Jetson TX1 running Ubuntu 16.04 operational system. The board employs an SOC design that incorporates a 64-bit quad-core ARM A57 processor. The cores are capable of frequency scaling from 0.1 GHz to 1.73 GHz with steps of 0.1 GHz. The CPUs share a 2 MB L2 cache and a 4 GB LPDDR4 Memory.

The load generator for Memcached are running on an AMD Ryzen 5 2600X Processor with 6 cores at 4.25 GHz with a shared Last Level Cache of 19 MB, and a 16 GB DRAM. The Jetson board and the machine hosting the load generator are connected using a 10 GB ASUS XG-C100C.

4.1.2 Benchmarks

As a starting point we selected Cloudsuite's [1] load generator for Memcached, an open-loop load generator, with uniform distribution. After performing a number of experiments, we found that Mutilate [7], a closed-loop load generator, was producing more stable results on our setup. As illustrated in figure 4.1, it deviates 3.72% from the sniffer at high load, whereas Cloudsuite overestimates more as the load increases, up to 32.4% at high load.

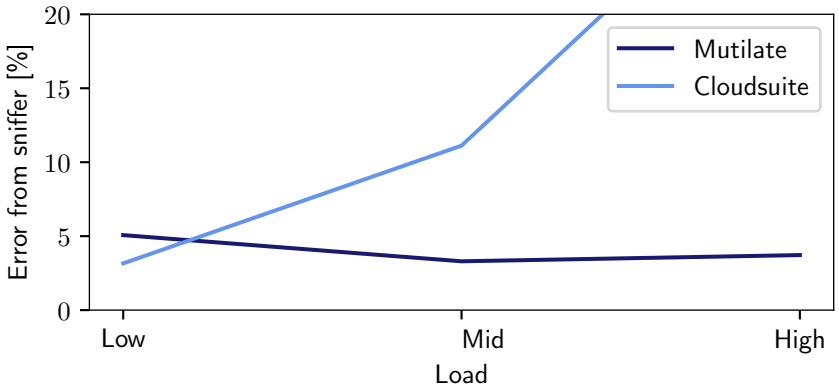


Figure 4.1: Comparison of the load generators' deviation to the sniffer

On the server, Memcached is running four server threads, equal to the number of cores on the Jetson TX1 board. We dedicate 2 GB of RAM, and set an upper bound of 5000 connections.

Due to limited access to hardware, our setup features a single-client configuration. According to Mutilate's recommendations, the number of worker threads should not exceed hardware cores/threads, and as the client is operating on a hexacore CPU with 12 threads, we choose to spawn 8 worker threads for Mutilate, leaving 4 threads to other background processes, like linux and the task manager. Following the guidelines from Mutilate, we are using 16 connections per worker thread. It is recommended establishing on the order of 100 connections per server thread, but our hardware limits disallows us to reach this number. We consider 128 connections in total sufficient for our experiments. For the distribution of key and value size, as well as inter-arrival time distribution, we chose to use Mutilate's built-in Facebook distribution.

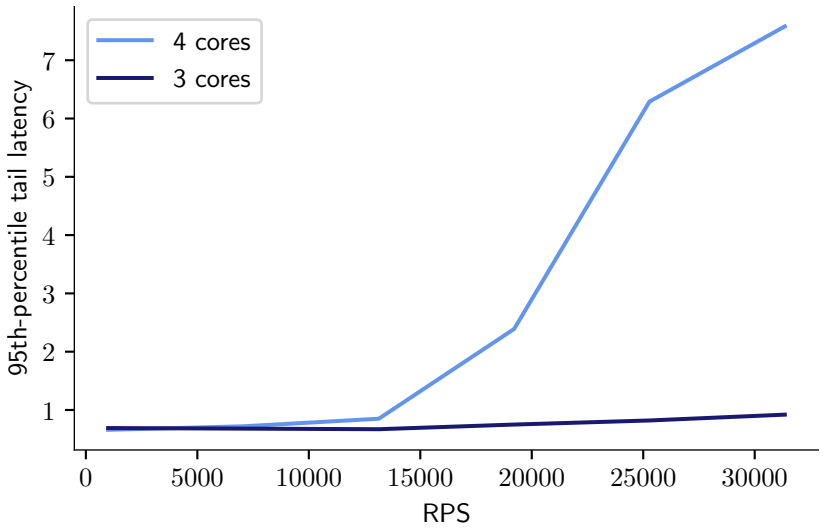


Figure 4.2: Latency for three and four cores

To simulate a time varying load, the load is generated as a step-wise monotonic function. The load difference between every load change is a constant change factor. We decide this to be 20% of the load range. The load between every load change is constant. The function starts at a minimum load and increases by the change factor every interval until maximum load is reached. One iteration is from 0% to 100%, and the same steps in reverse. Each load step is held 50 seconds.

When performing an experiment on 100% load, we discover a severe degradation of 724% on 95^{th}-percentile tail latency when running on all four cores compared to three, and an average degradation of 273% for all load steps in the benchmark at maximum DVFS settings. This can be seen in figure 4.2. We suspected this to be due to interference with the Linux kernel, as well as other unrelated software running in the background. The degradation is measured after increasing the priority of Memcached and bringing the count of background processes to a minimum. As a consequence, we decide to exclude the use of four simultaneous cores, leaving the maximum core setting to three cores.

4.1.3 Tail Latency

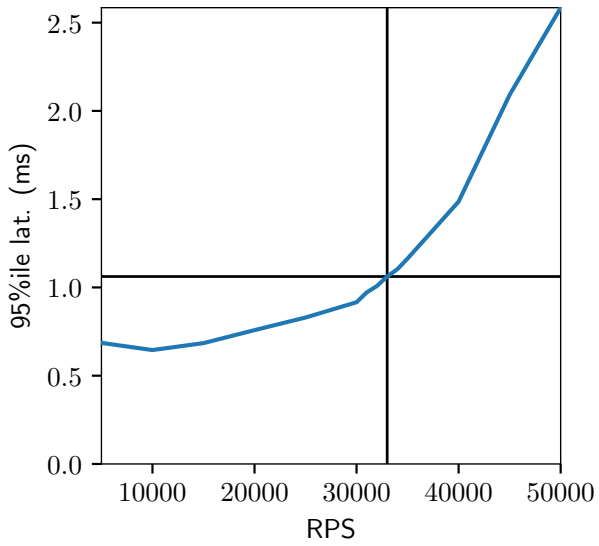


Figure 4.3: Tail latency and load at the knee

Figure 4.3 presents how the 95th percentile tail latency reacts to increasing load levels for Memcached. We use this to quantify the maximum input load the server can sustain while meeting QoS-target. The tail latency is increasing more rapidly after exceeding a load threshold. To quantify this threshold we use the *kneedle algorithm* [39] to locate the knee, marked as the intersection of the black lines in the figure. We set

the QoS-target to the latency at the knee, and the maximum load as 95% of the load at the knee, because the latency is unstable at 100%, as the system cannot handle more.

Our tests show that the knee is located at 33000 RPS and 1.1 ms. Hence, maximum load during our experiments is set to be 31350 RPS, and QoS-target is 1.1 ms.

4.1.4 Energy Measurements

The power consumption of the board is obtained by reading the INA3221 register. The register reports separately the power consumption of the GPU and CPU. The power consumption of the GPU is not taken into account, as it is not used in our experiments.

4.1.5 Prediction Model Evaluation

We show the accuracy of the prediction models by using: the coefficient of determination, R-squared (R^2), and the Percentage Absolute Average Error (PAAE) [18]. For a given prediction function $y = f(x)$, R^2 determines how much the total variation of Y (dependent variable) is due to X (independent variable). In other words, it is $1 - Z$, where Z is the ratio of the residual sum of squares to the total sum of squares, and is represented as:

$$R^2 = 1 - \frac{\sum(Y_{actual} - Y_{predicted})^2}{\sum(Y_{actual} - Y_{mean})^2} \tag{4.1}$$

PAAE is calculated as the average of the percentage absolute error for each sample predicted, as shown in equation 4.2, reproduced from [40]. N is the total number of data points, and the PAAE is the average of the absolute percentage deviation between the estimated value, $QoS_{predicted,i}$, from the actual value, $QoS_{actual,i}$.

$$PAAE = \frac{1}{N}\sum_{i=0}^{N}\frac{|QoS_{predicted,i} - QoS_{actual,i}|}{QoS_{actual,i}} \tag{4.2}$$

4.1.6 Evaluation Metrics

We evaluate Memcached using two metrics: *QoS-guarantee* and *QoS-tardiness*. QoS-guarantee is defined as the percentage of measured QoS samples that met the QoS-target. QoS-tardiness is defined as the ratio of measured QoS to the QoS-target, and it determines how intense the violation was. A QoS violation has occurred if the tardiness is above 1.

4.1.7 Machine Learning Model Parameters

In this section, we describe the parameters used for RF, SVM and RL. As a thumb rule, we use a randomly generated seed for all experiments.

Random Forest — When running RF alongside the load generator, we see an interference that leads to a drop in RPS of 10-15% at higher numbers of tree estimators. In addition, at 200 tree estimators, the estimations do not finish within the one second interval. We choose 20 tree estimators, as the overhead is minimal and the RPS close to target.

Support Vector Machine — For SVM we choose the parameters empirically. After testing various combinations, the following numbers obtained best results: The normalization factor, γ, is set to 1×10^{-4}, the C value is set to 100, and ϵ is set to 0.05.

Reinforcement learning — We set the parameters of RL equal to those in Hipster. This is a learning rate, α, of 0.1, a discount rate, γ, of 0.6, and ϵ of 0.05. As we do not measure lower latencies than 0.6 ms, we set the lower bound to 0.5 ms to add some margin. We allow for 10 actions, spanning from 0.5 to 1.1.

Offline models, like SVM and RF, require offline, labeled data for training. We generate this data set by running one iteration of the benchmark at all combinations of the three core mappings and 17 frequency steps, totalling at 51 configurations. Each load step is held for 50 s, and is sampled each second. As a result, the entire, offline training set consists of 30600 samples.

4.1.8 Baseline Comparisons

We compare our work against a static baseline, the Linux scheduler and two state-of-the-art solutions: Hipster [8], and Heracles [6].

Static All resources are allocated to the latency-critical workload at the highest DVFS setting.

Linux Both DVFS and core configuration is freely decided by Linux. The configuration is read out and sampled once every second. The active governor is *ondemand*.

Hipster is a hybrid reinforcement learning (RL) algorithm that combines heuristics with RL to determine mapping decisions based on the current load of the workload. The heuristic explored by Hipster is a state-machine based algorithm that orders the mapping configuration (cores and DVFS) in increasing order of power efficiency. A transition between states occurs when the tail latency is too close or too far away from the target. This heuristic enables Hipster to speed-up the learning and avoid using mapping decisions that may violate the QoS-target. The current load is quantized into multiple buckets as part of the state. The action for Hipster is a mapping configuration

for the latency-critical workload. We set the learning rate to 0.6 , discount factor to 0.9 , the bucket size to 4 % and the learning phase to one iteration in the benchmark (500 s).

To determine the order of the mapping configurations by power efficiency, we order by instructions per second per watt (IPS/W). We run a stress microbenchmark while counting instructions per second using *perf* and measuring power as useful energy consumption, removing the idle power, for each core/DVFS configuration. As an experiment, we also test ordering the configurations only by power consumption, as this results in the configuration with highest throughput last, as opposed to third to last as for IPS/W.

.

Heracles Heracles is a heuristic mapper that aims to meet the tail latency of latency-critical workload. Heracles maintains three levels of the feedback controllers: (1) main (2) core and memory (3) power controller. The main controller is polled every 15 second and is responsible for suspending non-latency-critical jobs, if the latency-critical workload either violates the QoS-target or if the load is higher than 85%. In such cases, all resources are given to the latency-critical workload for a period of 5 min. The core and memory controller is polled every 2 s, and is responsible allocating cores and memory resources to the workload. If the tail latency is at 80% of the QoS-target or if the measured memory bandwidth has increased, then the latency-critical workload is allocated an additional core with an increase to the LLC allocation using the Intel cache allocation technology (CAT) [41].[1] In all other cases, a core is de-allocated from the latency-critical workload. The power controller is polled every 2 s, and is responsible for decreasing the DVFS setting when the current power is at 90% of maximum measured power consumption.

4.2 Results

In this section we present the results obtained from the evaluation of the implemented machine learning models, SVM, RF and RL. We also present the effectiveness of Heimdall when deployed with Memcached and compare the performance results to the baseline solutions presented in 4.1.8.

4.2.1 Sniffer

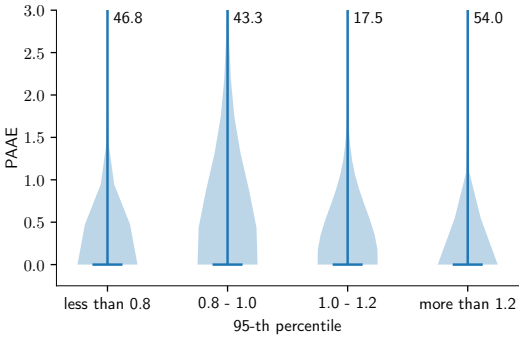
From figure 2.4 we see that the sniffer eliminates the queuing latency of Memcached, and as figure 4.1 illustrates, it shows improvements to both Mutilate and Cloudsuite of

[1] The Nvidia Jetson board does not offer CAT, and therefore was not used in our experiments.

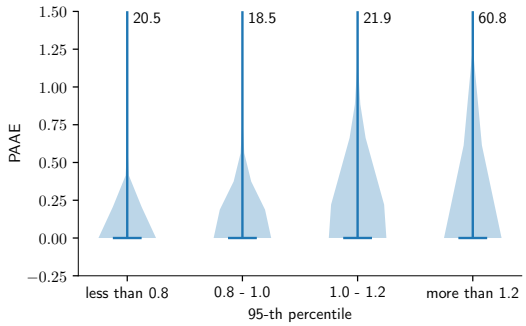
up to 3.72% and 32.4% respectively. We also successfully capture and obtain metrics like RPS and packet length for an additional workload; web search. However, due to time constraints, web search is not used in the evaluation of machine learning models and task managers.

4.2.2 Machine Learning Models

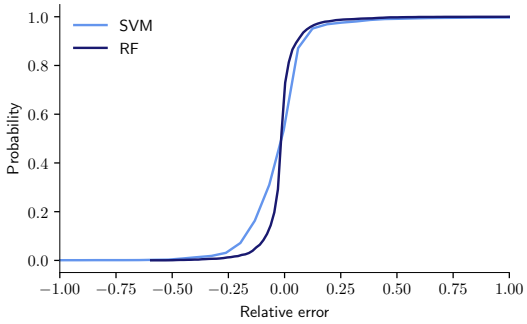
Offline learning



(a) Violin plot of PAAE on different tail latencies for SVM



(b) Violin plot of PAAE on different tail latencies for RF



(c) Cumulative distribution function (CDF) of tail latency prediction error for SVM and RF

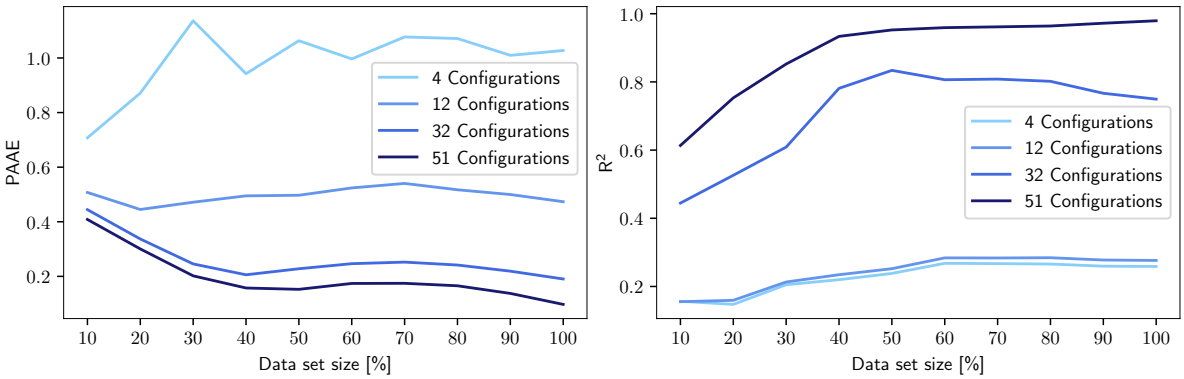
Figure 4.4: Violin plot and CDF of estimation error

Figure 4.4 shows the precision of the prediction models, RF and SVM. In this experiment we use 70% of the offline data set for training, and the remaining 30% to test the performance of the model. For each of the 51 core/DVFS configurations, we select 30% of the samples at random to append to the test set. The remaining 70% is put into the training set.

The lower subfigure (c) shows the prediction error as a cumulative distribution function. The x-axis represents the relative error, where 1.0 corresponds to a 100%

deviation. We observe that both models have a tendency of underestimating tail latency, as both distributions are slightly shifted to the left of 0. On the other hand, only around 34% of the samples are within the QoS-target, and the remaining samples are spread up to 700 ms. We measure on average for the lowest configuration on maximum RPS, a tail latency of more than three hundred times the QoS-target. Our hypothesis is that this is the reason why the largest outliers for RF are overestimations. This is also supported by subfigure (b), showing that the largest outlier of the violating samples are more than twice of the non-violating samples.

These graphs show that it is possible to estimate tail latency with low error rate using the prediction models. As it comes clear that RF performs slightly better than SVM in this experiment, we expand our analysis with emphasis on RF.



(a) PAAE of RF for different configuration **(b)** R^2 score of RF on different configurations

Figure 4.5: Performance of different sets of configurations

Figure 4.5 shows how training on smaller fractions of the data set impacts the prediction accuracy using RF. Each line corresponds to a given number of core/DVFS configurations, and the x-axis represents the fraction of the data set for those core/DVFS configurations being used for training. After training, the model is tested on the composition of 100% of the data set for all 51 configurations. We select the configurations for training randomly, with the only requirement that all unique core mappings are represented at least once. The data set fraction is chosen deterministically, where the data is chosen sequentially for each configuration, from the beginning up to the percentage desired.

The left subplot (a) demonstrates how the configuration count and data set fraction used during testing affects the PAAE. Using four configurations performs poorly, with a PAAE of around 1. However, already at 12 configurations it is stabilizing, and the PAAE is virtually unaffected by the data set size. The difference in performance between using all 51 configurations and only 32 configurations is minimal.

The right subplot (b) illustrates the effect on R^2 for configuration count and data

set size. We observe that R^2 increases up to a data set size of around 50%, but then flattens out. This might be explained by the fact that the data set consists of the load steps from the benchmark, first increasing load, then decreasing down the same load steps. In other words, from 50% of the data set size and out, the same load steps are revisited.

Figure 4.6 extracts the line with maximum configurations in figure 4.5. It clearly illustrates that PAAE is reduced while R^2 increases, and that the benefit of increasing data set size diminishes after passing 40%. From this we argue that we can use less data to make predictions with high accuracy.

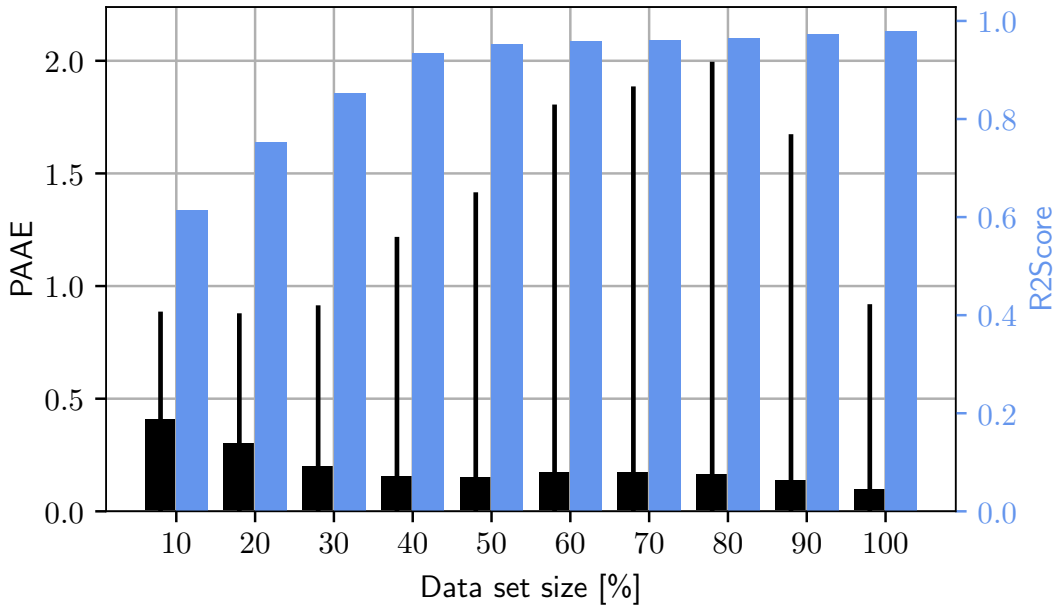


Figure 4.6: PAAE and R^2 score for RF

Online learning

RL is the only prediction model used in this work producing discrete estimates. Evaluating this model will as a consequence be slightly different from the other models. PAAE would be an unfair metric, as RL is incapable of measuring higher values than QoS-target, and the actual latency may be several hundred times higher.

We will therefore evaluate RL from correctness and PAAE, where the PAAE is calculated only for the non-violating samples. Correctness quantifies the proportion of estimates where the actual value and the estimated value reside in the same estimation bucket, for both violating and non-violating samples.

After only one iteration of the benchmark starting from a uniform model, RL has a correctness of 71.7% and a PAAE 0.09. When the model is trained on one iteration of the benchmark in advance, the correctness is increased by 10.3% to a total of 79.1%.

For the same experiment the PAAE nearly cut one third, to 0.06. Training on the same data set more times in advance does not benefit significantly, as training on the same data set hundred times in advance only increases correctness to 80.2% and PAAE to 0.05.

The advantage of using such online models is that it requires no data before hand, and adjusts itself to changes, as opposed to the offline models training on data before starting, and never being able to adapt if the work load would change. The disadvantage, however, is that online learning models are initially imprecise.

For a closed-loop load generator like Mutilate, the RPS generated is limited by the responsiveness of the server. In our case, having 128 connections and a maximum load of 31350 RPS, the average response time must be under 4 ms to maintain the request rate. Under lower energy core/DVFS configurations, the load generator fails to generate requests at that rate, and the RPS deviates from target RPS.

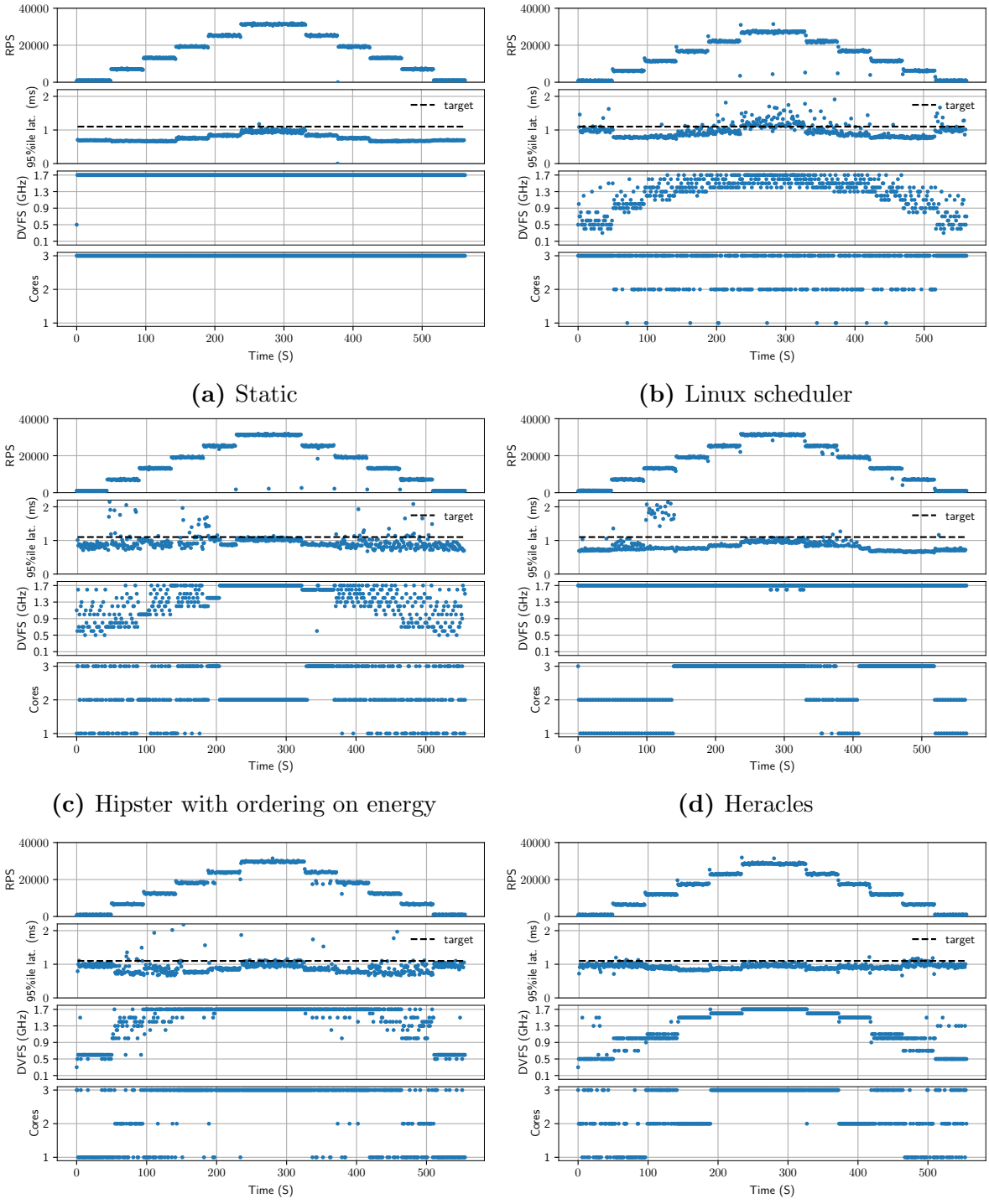
Due to this misbehavior, states might not be consistent across actions. A misprediction by our RL model might, if it underestimates the tail latency of a lower configuration, alter the state. This has two implications: RL is learning to estimate on wrong states. Second, utilizing RL for an exhaustive search on core/DVFS configurations is not valid, as the state before and after the configuration is updated may differ.

With this in mind, we do not find it reasonable to experiment with RL beyond latency estimations with our setup, and it is not used for scheduling.

4.2.3 Task Manager

In this subsection, we evaluate the effectiveness of the task managers when using the different machine learning models, and how they perform compared to the baselines.

Figure 4.7 shows the results of the different task manager schemes when managing Memcached. Subfigure (a) to (d) show the performance of the baselines we are comparing against: Linux scheduler, static, Hipster and Heracles. Subfigure (e) and (f) shows the results of Heimdall using SVM and RF respectively. The x-axis in each subfigure represents the time of the experiment. The first row in each subfigure shows the load represented as RPS. The second row presents the 95th percentile latency. This row also has a black, dashed line which marks the QoS-target. The third row shows the DVFS setting, while the last row presents the core mapping. Figure 4.8 shows the results of task management with Hipster, ordering by IPS/W.



(a) Static

(b) Linux scheduler

(c) Hipster with ordering on energy

(d) Heracles

(e) SVM

(f) Random Forest

Figure 4.7: Benchmark results for task managers

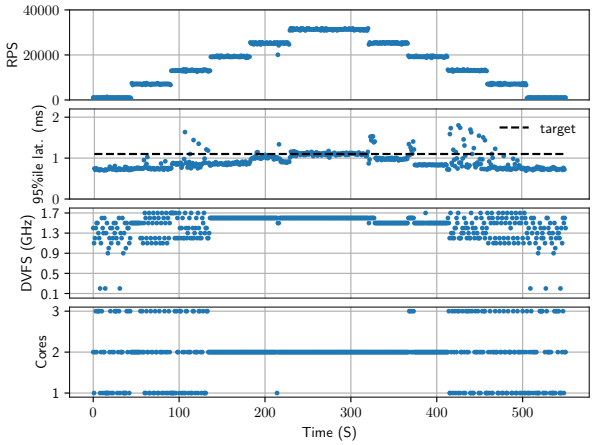


Figure 4.8: Benchmark results for Hipster with ordering on IPS/W

Table 4.1 presents the QoS-guarantee, QoS-tardiness and energy reduction for the different task managers we tested: Static, Linux scheduler, Hipster, Heracles, Heimdall with SVM and Heimdall with RF. The QoS-tardiness values in the table are the average of the QoS-tardiness only including violating samples. Because static utilizes all resources at maximum DVFS independent of load, it will be the most reliable to meet the QoS-target, but also most power hungry. Therefore we use static as a baseline and report the energy consumption of all task managers compared to static.

	QoS Guarantee	*QoS Tardiness*	*Energy Reduction*
Static	99.8%	1.1	-
Linux	78.1%	1.2	21.4%
Hipster ordered by energy	87.8%	2.1	20.6%
Hipster ordered by IPS/W	84.0%	1.7	22.1%
Heracles	92.1%	3.0	7.2%
Heimdall with SVM	89.9%	4.7	7.1%
Heimdall with RF	97.7%	1.1	22.3%

Table 4.1: Summary of QoS guarantees, tardiness and energy savings for Memcached

Static has a high QoS-guarantee, and the QoS-tardiness is low. This is due to excessive resource usage, and the power consumption is consequently the highest of all the task schedulers.

Linux scheduler dynamically schedules after the current load. This scheduler is rather deterministic, and after running multiple times, we see similar results. By making mapping decision only based on CPU utilization, the QoS-guarantee for this task manager is the lowest of all task schedulers tested in this work, as it does not take

into consideration pressure on other resources like memory or network. This is more evident during maximum load, as the Linux scheduler attempts to run at lower DVFS, when it is necessary to run at maximum DVFS to meet the QoS-target. Furthermore, the context switch time of the Linux scheduler is 10 ms, one hundredth of Heimdall and Hipster, allowing it to respond to sub second load variations, and thus reduce the QoS-tardiness. The inconvenience is the overhead, and affects latency negatively.

Hipster The figure shows that Hipster ordered by energy exhibits a ping-ponging effect for small variations in load, that causes violations to the QoS. Besides, the state space expands rapidly as additional configurations are added, raising a memory problem. Figure 4.8 shows Hipster ordered by IPS/W. This approach saves slightly more power, but at the cost of the QoS-guarantee. During the heuristic training phase, the static order generated from IPS/W on the Jetson TX1 board slightly favors more energy efficient configurations for higher loads, compared to when the static order is generated from energy consumption.

Heracles The core and memory controller periodically increases or decreases the number of cores based only on tail latency. This leads to a high number of task migrations compared to Hipster. The main controller is polled every 15 seconds and if the QoS-target is violated it adjusts the number of cores and DVFS to maximum for 2 minutes. We can see that a suspension period is triggered after 140 seconds because of QoS-violations. The mapping decisions and DVFS during these periods are the same as for static, resulting in an absence of energy savings. It is also important to notice that the DVFS is almost exclusively set to maximum, as it is only reduced if the energy measured is close to the maximum measured power consumption.

The QoS-guarantee is the third highest of all task schedulers tested in this work. However, a significant amount of QoS-violations take place between 100 and 130 seconds, after a change in load. It suffers from a course-grained action space as the DVFS is constant and the only option is increasing or decreasing cores by one. From the 100 second mark running two cores results in a latency lower than the threshold of 80% of QoS-target, while running one core results in a QoS-violation. This causes an oscillation between the two core mappings, visible as a ping-ponging effect. This persists until a suspension is forced by the main controller.

Heimdall with SVM overestimates the latency at higher load, and in response, the DVFS and core configuration are maximized at an early, suboptimal point, after around 100 seconds. Using this model, Heimdall misses out on energy savings, and only reduce energy consumption by 7.1% compared to static. Despite running mostly on maximum configuration from 100 seconds, the prediction model gives occasionally inconsistent predictions with high variability in the transition from low to high load, load steps three to five, that cause a significant drop in QoS-guarantee.

Heimdall with RF produces more precise estimates, that in turn reflects the performance of different core/DVFS configurations with respect to the current load better than SVM. Accordingly, Heimdall is able to select the lowest performing configurations that meet the QoS-target, minimizing energy consumption while maintaining a high QoS-guarantee.

4.2.4 Discussion

Out of the six various task managers we have tested, Heimdall with RF performs better. With the parameters we chose for our experiments, RF provides more precise estimates than SVM, which causes Heimdall to perform better with RF than SVM. Finer tuning of parameters would possibly improve SVM, but our efforts in methods like 5-fold cross validation [42] has not yielded improvements. RF could benefit from being placed on dedicated hardware, as interference with the load generator would be eliminated.

Heracles has, despite poor energy reduction, an advantage to Hipster and Heimdall by not requiring neither upfront data, nor any up front training period. Hipster requires a profiling of core/DVFS configurations in advance, and Heimdall requires a training data set of various core/DVFS configurations. From the results in section 4.2.2, we show that this data set does not require data from all possible configurations to produce precise estimations, but rather that an acceptable level of precision may come from a smaller data set.

Chapter 5

Conclusion

In this work we have developed Heimdall, a task manager incorporating a packet sniffer. We have shown that the sniffer we built gives estimates as good or better than those provided by the load generators Mutilate and Cloudsuite. We also achieve reasonable precision on estimated latency predictions from the machine learning models SVM, RF and RL. Using these estimates, our results for Heimdall show that, we improve over the Linux scheduler, Hipster and Heracles in meeting QoS by 25%, 11.2% and 6.1%, respectively, while reducing energy consumption by 1.2%, 2% and 16%, respectively.

Bibliography

[1] *Cloudsuite*. URL: `http://cloudsuite.ch`. (accessed: 28.05.19).

[2] NRDC. "America's Data Centers Are Wasting Huge Amount of Energy". In: (2014).

[3] David Lo, Liqun Cheng, Rama Govindaraju, Luiz André Barroso, and Christos Kozyrakis. "Towards energy proportionality for large-scale latency-critical workloads". In: *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. IEEE. 2014, pp. 301–312.

[4] Jeffrey Dean and Luiz André Barroso. "The tail at scale". In: *Communications of the ACM* 56.2 (2013), pp. 74–80.

[5] Eric Schurman and Jake Brutlag. "The user and business impact of server delays, additional bytes, and http chunking in web search". In: *Velocity Web Performance and Operations Conference*. 2009.

[6] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. "Heracles: Improving resource efficiency at scale". In: *ACM SIGARCH Computer Architecture News*. Vol. 43. 3. ACM. 2015, pp. 450–462.

[7] Jacob Leverich et al. "Reconciling High Server Utilization and Sub-millisecond Quality-of-service". In: EuroSys 2014.

[8] Rajiv Nishtala, Paul Carpenter, Vinicius Petrucci, and Xavier Martorell. "Hipster: Hybrid task manager for latency-critical cloud workloads". In: *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2017, pp. 409–420.

[9] Chang-Hong Hsu, Yunqi Zhang, Michael A Laurenzano, David Meisner, Thomas Wenisch, Jason Mars, Lingjia Tang, and Ronald G Dreslinski. "Adrenaline: Pinpointing and reining in tail queries with quick voltage boosting". In: *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2015, pp. 271–282.

[10] Chang-Hong Hsu, Qingyuan Deng, Jason Mars, and Lingjia Tang. "SmoothOperator: reducing power fragmentation and improving power utilization in large-scale datacenters". In: *ACM SIGPLAN Notices*. Vol. 53. 2. ACM. 2018, pp. 535–548.

[11] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. "Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations". In: *Proceedings of the 44th annual IEEE/ACM International Symposium on Microarchitecture*. ACM. 2011, pp. 248–259.

[12] Balajee Vamanan, Hamza Bin Sohail, Jahangir Hasan, and TN Vijaykumar. "Timetrader: Exploiting latency tail to save datacenter energy for online search". In: *Proceedings of the 48th International Symposium on Microarchitecture*. ACM. 2015, pp. 585–597.

[13] Ripal Nathuji, Aman Kansal, and Alireza Ghaffarkhah. "Q-clouds: managing performance interference effects for qos-aware clouds". In: *Proceedings of the 5th European conference on Computer systems*. ACM. 2010, pp. 237–250.

[14] Harshad Kasture, Davide B Bartolini, Nathan Beckmann, and Daniel Sanchez. "Rubik: Fast analytical power management for latency-critical systems". In: *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. 2015, pp. 598–610.

[15] *Memcached - a distributed memory object caching system*. URL: `https://memcached.org/`. (accessed: 28.05.19).

[16] Jacob Leverich and Christos Kozyrakis. "Reconciling high server utilization and sub-millisecond quality-of-service". In: *Proceedings of the Ninth European Conference on Computer Systems*. ACM. 2014, p. 4.

[17] Simon Holmbacka. "Energy aware software for many-core systems". In: 2015.

[18] Sergey Blagodurov et al. "A case for NUMA-aware contention management on multicore systems". In: PACT 2010.

[19] S. Zhuravlev, J. C. Saez, S. Blagodurov, A. Fedorova, and M. Prieto. "Survey of Energy-Cognizant Scheduling Techniques". In: *IEEE Transactions on Parallel and Distributed Systems* 24.7 (July 2013), pp. 1447–1464. ISSN: 1045-9219. DOI: `10.1109/TPDS.2012.20`.

[20] V. Petrucci, M. A. Laurenzano, J. Doherty, Y. Zhang, D. Mossé, J. Mars, and L. Tang. "Octopus-Man: QoS-driven task management for heterogeneous multicores in warehouse-scale computers". In: *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. Feb. 2015, pp. 246–258. DOI: `10.1109/HPCA.2015.7056037`.

[21] Bernhard E Boser et al. “A Training Algorithm for Optimal Margin Classifiers”. In: COLT 1992.

[22] Leo Breiman. “Random Forests”. In: *Mach. Learn. 2001* ().

[23] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, et al. “Scaling memcache at facebook”. In: *Presented as part of the 10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*. 2013, pp. 385–398.

[24] Twitter Inc. *Twemcache*. URL: `https://github.com/twitter/twemcache`. (accessed: 29.05.2019).

[25] *Protocols*. URL: `https://github.com/memcached/memcached/wiki/Protocols`. (accessed: 11.06.19).

[26] *Web Search*. URL: `https://github.com/parsa-epfl/cloudsuite/blob/master/docs/benchmarks/web-search.md`. (accessed: 11.06.19).

[27] *Apache Solr*. URL: `https://lucene.apache.org/solr/`. (accessed: 11.06.19).

[28] *Transfer-Encoding*. URL: `https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Transfer-Encoding`. (accessed: 11.06.19).

[29] Yunqi Zhang, David Meisner, Jason Mars, and Lingjia Tang. “Treadmill: Attributing the source of tail latency through precise load testing and statistical inference”. In: *ACM SIGARCH Computer Architecture News* 44.3 (2016), pp. 456–468.

[30] *TCPDUMP & LIBPCAP*. URL: `https://www.tcpdump.org/`. (accessed: 17.04.19).

[31] *Pcap4J*. URL: `https://www.pcap4j.org/`. (accessed: 17.04.19).

[32] *pcap: A system-independent interface for user-level packet capture*. URL: `http://hackage.haskell.org/package/pcap`. (accessed: 17.04.19).

[33] *python-libpcap*. URL: `https://sourceforge.net/projects/pylibpcap/`. (accessed: 17.04.19).

[34] *Wireshark*. URL: `https://www.wireshark.org/`. (accessed:29.05.19).

[35] *Transmission Control Protocol*. URL: `https://tools.ietf.org/html/rfc793`. (accessed: 21.04.19).

[36] Harald Øverby. *TCP*. URL: `https://snl.no/TCP`. (accessed: 21.04.19).

[37] Christina Delimitrou and Christos Kozyrakis. “Amdahl’s law for tail latency”. In: *Communications of the ACM* 61.8 (2018), pp. 65–72.

[38] *Scikit-learn: Machine Learning in Python*. URL: https://scikit-learn.org. (accessed: May 2019).

[39] Ville Satopaa, Jeannie Albrecht, David Irwin, and Barath Raghavan. "Finding a" kneedle" in a haystack: Detecting knee points in system behavior". In: *2011 31st International Conference on Distributed Computing Systems Workshops*. IEEE. 2011, pp. 166–171.

[40] Kishore Kumar Pusukuri, David Vengerov, and Alexandra Fedorova. "A methodology for developing simple and robust power models using performance monitoring events". In: *proceedings of WIOSCA* 9 (2009).

[41] Nguyen, Khang T. *Software Enabling for Cache Allocation Technology in the Intel Xeon Processor E5 v4 Family*. URL: https://software.intel.com/en-us/articles/software-enabling-for-cache-allocation-technology. (accessed: January 2019).

[42] Chih-Wei Hsu, Chih-Chung Chang, Chih-Jen Lin, et al. "A practical guide to support vector classification". In: (2003).